Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра интеллектуальных информационных технологий

Оплачко Николай Алексеевич

Оптимизация вычислений нейронных сетей на графических процессорах для мобильных устройств

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
д.ф-м.н., профессор, академик РАН
А.И. Аветисян

Москва, 2021

Оглавление

1. Введение

В последние годы нейронные сети (НС) приобрели большую популярность в применениях машинного обучения к различным прикладным задачам. Отчасти это обусловлено приростом вычислительных мощностей графических процессоров (ГП): современные НС часто обучают и запускают с использованием нескольких ГП, вычисления происходят на облачных серверах и кластерах.

В приложениях на мобильных устройствах с использованием НС иногда возникает потребность вычисления предсказаний НС непосредственно на самих мобильных устройствах. Причин этому несколько, в частности: защита данных пользователя, возможность использования приложения без доступа к сети Интернет, сокращение затрат на сервера, уменьшение временной задержки на получение предсказания НС.

Вместо использования центрального процессора мобильного устройства для вычисления предсказаний НС более эффективно вычисление на его ГП, поскольку многие операции в НС можно разбить на множество маленьких задач, решаемых независимо друг от друга, что вполне соответствует архитектурным особенностям ГП.

При решении задач, связанных с классификацией и обработкой изображений и видеопоследовательностей, активно используются разновидности идеи сверточных нейронных сетей (СНС), предложенной в работе [1]. Также СНС оказались эффективными и в решении задач из области обработки естественного языка [2], [3]. На примере сети ResNeXt [4] было показано, что архитектурная реализация сверточных слоев в СНС с помощью групповых сверток, изначально предложенных в работе [5], улучшает точность предсказания, а также вычислительно более эффективна по сравнению с классическими свертками.

Операции двумерной свертки в СНС вычислительно затратны и возникает естественное желание добиться эффективности их вычисления. Для их ускорения при фиксированном размере ядра используются вариации алгоритма вычисления сверток Винограда [6] — это позволяет ускорить вычисление операции свертки в несколько раз.

В силу ограниченности ресурсов мобильного устройства также есть проблемы потребления памяти и электрической энергии. Особенности обра-

щений в память ГП влияют на время их выполнения, что тоже сказывается на временных затратах.

Основной вклад данной работы заключается в следующем:

- разработан метод оптимизации, позволяющий сокращать объем используемой памяти и общее время исполнения;
- предложенный метод реализован в программно-аппаратной части библиотеки искусственного интеллекта MindSpore Lite[1] на языке C++ с использованием интерфейса прикладного программирования Vulkan;
- произведено эмпирическое исследование предложенного метода и его сравнение с базовой реализацией и подходами из работ [7], [8].

Настоящая работа организована следующим образом. В разделе 2 формально описываются задача оптимизации памяти и задача параллельных вычислений. Обзор существующих подходов к их решению приводится в разделе 3. В разделе 4 описывается предложенный метод. Экспериментальное исследование метода и сравнение с другими подходами приводится в разделе 5.

[1]Кодовая база `MindSpore`: `https://github.com/mindspore-ai/mindspore`.

2. Постановка задачи

Пусть $\mathbb{R}$ — множество вещественных чисел, будем называть *тензорами* многомерные массивы с элементами из $\mathbb{R}$, $\mathbb{T} = \bigcup\limits_{n=1}^{\infty} \bigcup\limits_{\mathbb{T}_1,\ldots,\mathbb{T}_n \in \{\mathbb{R}^s | s \in \mathbb{N}^+\}} \mathbb{T}_1 \times \cdots \times \mathbb{T}_n$ — множество всевозможных кортежей из тензоров.

Определение 2.1. *Ациклическим графом вычислений (АГВ)* назовем тройку $\langle G(V,E), \mathbf{op}, \mathbf{t} \rangle$, где $G(V,E)$ — ориентированный ациклический граф, в котором:

- $\mathbf{op} : V \to (\mathbb{T} \to \mathbb{T})$ — сопоставление вершинам операций, которые отображают набор входящих в вершину тензоров в набор выходящих.
- $\mathbf{t} : E \to \bigcup\limits_{s \in \mathbb{N}^+} \mathbb{R}^s$ — каждому ребру соответствует некоторый тензор, причем разные ребра могут соответствовать одному и тому же тензору тогда и только тогда, когда эти ребра исходят из одной вершины.

Вычисление предсказаний НС можно представить в виде АГВ, пример такого представления приведен на Рис. 1. Данная работа сфокусирована на улучшении программно-аппаратной части вычисления предсказаний НС, без рассмотрения оптимизации архитектуры самой НС или процесса ее обучения. Целью работы является исследование существующих методов оптимизации объема используемой памяти и затраченного времени на вычисление предсказания НС на уровне АГВ, а также разработка и реализация метода, позволяющего одновременно оптимизировать и затраченное время, и объем используемой памяти.

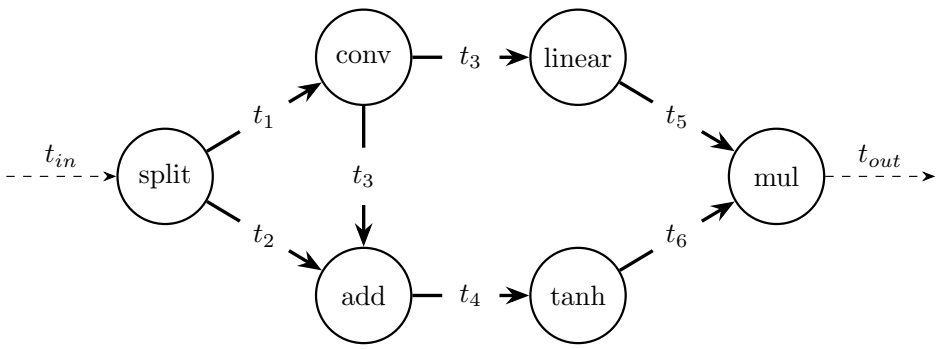


Рис. 1: Пример ациклического графа вычислений
Пунктирными линиями изображены входной и выходной тензор, которые присутствуют в вычислениях, но отсутствуют в самом графе.

2.1. Задача оптимизации памяти

Параметры НС, представляющие собой тензоры, можно разделить на четыре категории:

- Константные, так же веса НС
- Входные — то, что подается на вход НС
- Выходные — предсказание НС
- Промежуточные — не относящиеся к трем остальным категориям

Входные и константные тензоры копируются из оперативной памяти в память ГП перед вычислением сети, выходные — копируются из памяти ГП в оперативную память после произведенных вычислений. Для промежуточных же тензоров память может быть использована повторно, если это не нарушает корректности производимых вычислений.

В настоящей работе мы рассматриваем задачу предварительного распределения памяти, то есть размеры промежуточных тензоров НС заранее известны. Выделение блока памяти производится указанием для промежуточных тензоров сдвигов на соответствующие им ячейки памяти, начиная с которых тензоры могут быть записаны или считаны, располагаясь как непрерывные блоки. Выбор такого подхода работы с памятью обусловлен тем, что интерфейс прикладного программирования Vulkan, используемый для реализации программно-аппаратной части MindSpore Lite, предполагает резервирование приложением продолжительно используемой памяти, которую он может самостоятельно использовать.

Для соблюдения корректности результата последовательное вычисление НС необходимо производить в порядке любой топологической сортировки АГВ (ТС), т.е. для вычисления в данный момент некоторой операции необходимо и достаточно, чтобы все входные тензоры были вычислены.

Формулировка задачи оптимального распределения памяти

Обозначим множество тензоров, соответствующих ребрам в АГВ $\langle G(V, E), \mathbf{op}, \mathbf{t}\rangle$, как $T = \mathrm{im}(\mathbf{t}) = \left\{\mathbf{t}(e_1), \ldots, \mathbf{t}(e_{|E|})\right\}$. Для тензора $t \in T$ обозначим размер тензора, который он занимает в памяти, как $w_t \in \mathbb{N}$. Тогда задачу распределения памяти можно задать в виде следующей задачи оптимизации:

$$\begin{cases} \max\limits_{t \in T}(x_t + w_t) \to \min & (2.1а) \\ x_{t_1} + w_{t_1} \leqslant x_{t_2} \vee x_{t_2} + w_{t_2} \leqslant x_{t_1} & \\ \forall t_1, t_2 \in T : t_1 \text{ и } t_2 \text{ могут использоваться одновременно} & (2.1б) \\ x_t \in \mathbb{Z},\ x_t \geqslant 0 \qquad \forall t \in T & (2.1в) \end{cases}$$

где x_t — сдвиг в памяти ячейки начала размещения тензора t. Ограничение 2.1б запрещает тензорам использовать одну и ту же память, если может возникнуть конфликт, а 2.1в отражает то, что сдвиг в памяти обязан быть целым неотрицательным числом. При этих ограничениях требуется использовать как можно меньший объем памяти, этому соответствует выражение 2.1а.

2.2. Задача параллельных вычислений

Вычисление НС с помощью программы представляется в виде конвейера из команд, которые можно логически разбить на следующие типы:

1) Копирование блока из оперативной памяти в память ГП
2) Копирование блока из памяти ГП в оперативную память
3) Исполнение вычислительного *шейдера* (программы, выполняющейся параллельно несколькими блоками потоков ГП)
4) Барьер памяти вида запись/чтение между операциями типа 1 и операциями типа 3.
5) Барьер памяти вида запись/чтение между операциями типа 3 и операциями типа 3.
6) Барьер памяти вида запись/чтение между операциями типа 3 и операциями типа 2.

Перед началом интересующих нас вычислений выполняются команды типов 1 и 4, после — 6 и 2. Операции (вершины в АГВ) суть один или несколько вычислительных шейдеров, их вычисление относится к типу команд 3, а между вычислением вершин размещаются при необходимости барьеры памяти 5.

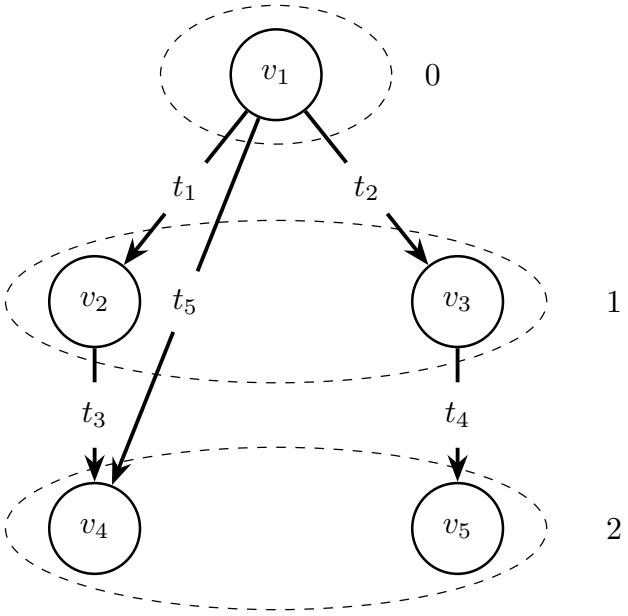


Рис. 2: Пример АГВ, в котором возможно параллельное вычисление некоторых вершин
Пунктирными линиями обведены подмножества
независящих друг от друга вершин

Под *базовой программной реализацией* вычислений НС с помощью ГП будем подразумевать следующее: вычисление операций происходит в порядке произвольной топологической сортировки АГВ; между каждой последовательной парой операций в конвейере располагается барьер памяти; для каждого тензора выделяется отдельный блок памяти, непересекающийся ни в какой момент времени с остальными.

При такой реализации возникает проблема избыточных синхронизаций, вызывающих простои в вычислениях. Сократить время простоев возможно, разбив АГВ на несколько блоков так, что внутри каждого блока вершины будут попарно *независимы* (не существует пути из одной в другую), и можно выбрать порядок вычисления этих блоков, при котором все зависимости будут соблюдаться (аналогично топологической сортировке вершин). Пример такого разбиения приведен на Рис. 2.

Задача параллельных вычислений состоит в том, чтобы найти эффективный способ получать разбиение АГВ на непересекающиеся подмножества попарно независимых вершин так, чтобы не было циклической зависимости этих подмножеств друг от друга. Также нужно исследовать применимость способа на практике.

3. Обзор литературы

3.1. Задача оптимизации памяти

На Рис. 1 приведен пример АГВ: можно заметить, что, например, выделенные блоки памяти для следующих пар тензоров:

- (t_2, t_3), (t_5, t_6) — не могут пересекаться независимо от выбора ТС
- (t_1, t_4), (t_1, t_5), (t_3, t_6) — могут пересекаться независимо от выбора ТС
- (t_2, t_4), (t_4, t_6) — могут пересекаться, если размерности соответствующих входов и выходов совпадают для операций **add** и **tanh**, поскольку они являются поэлементными операциями
- (t_3, t_4) — могут или не могут пересекаться в зависимости от ТС (необходимо ли хранить значение t_3 после вычисления операции **add**) и совпадения размерностей t_3 и t_4

На Рис. 3 для этого примера приведена одна из возможных топологических сортировок и помечены времена жизни для тензоров.

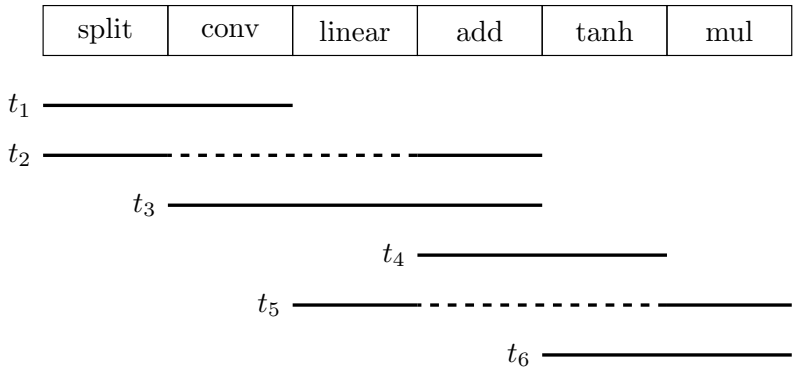


Рис. 3: Время жизни тензоров

Задача 2.1 является NP-полной в случае различных размеров тензоров и сводится, например, к задаче 3-разбиений [9]. Тем не менее возможен поиск приближенного решения эвристическими алгоритмами [10], [7].

В работе [10] задача 2.1 выражается как частный случай задачи упаковки прямоугольников (разместить прямоугольники в контейнер с заданной шириной и наименьшей высотой), решаемый эвристическим алгоритмом. В работе [7] для каждого тензора t АГВ вычисляется кратчайший отрезок вершин, включающий в себя все вершины, для которых t является входящим или выходящим ребром (назовем его *временем жизни* t); тензорам t_1 и t_2

разрешено использовать общую память тогда и только тогда, когда пересечение их времен жизни пусто. Далее описывается жадный алгоритм, в котором перебираются тензоры по убыванию их размера, и для каждого производится поиск наименьшего зазора в памяти, вмещающего его. Экспериментально показывается, что описанная ими стратегия достигает результатов не хуже (иногда лучше), чем эвристический метод из [10], на примере нескольких НС.

3.2. Задача параллельных вычислений

Для разбиения множества вершин АГВ на непересекающиеся подмножества независимых вершин можно использовать следующий подход, представляющий собой динамическое программирование на ациклическом графе (предложен, например, в [8]):

1) Рассмотрим вершины графа в порядке топологической сортировки:
   - если у очередной вершины v нет входящих ребер, инициализируем $maxdist(v) = 0$;
   - иначе, находим у вершины v предка w с наибольшим значением $maxdist(w)$, и затем инициализируем $maxdist(v) = maxdist(w) + 1$.
2) Пусть $D = \max\limits_{v \in V} maxdist(v)$. Построим для вершин V разбиение на подмножества: $\{L_0, \ldots, L_D\}$, где $L_i = \{v \mid maxdist(v) = i\}$.

Будем называть $L_i, i \in \overline{0, D}$ слоями АГВ. Очевидно, $\forall i, j \in \overline{0, D}\ i \neq j \rightarrow L_i \cap L_j = \varnothing$. Покажем, что в каждом множестве находятся только попарно независимые вершины.

Утверждение 3.1. $\forall i \in \overline{0, D}\ \forall v, u \in L_i$ v и u — независимы.

Доказательство. Допустим, что v и u не являются независимыми, то есть существует либо путь из v в u, либо из u в v. Но тогда длина наибольшего пути не может совпадать: в первом случае $maxdist(v) < maxdist(u)$, во втором $maxdist(u) < maxdist(v)$. Следовательно, если v и u принадлежат одному и тому же слою, то они независимы. □

Такой способ не гарантирует нахождения наибольших групп независимых вершин, однако является вычислительно эффективным и для представлений НС в виде АГВ находит приемлемое решение, что показано в разделе 5.

4. Предложенный метод

Предлагается для начала рассмотреть нижеизложенный способ, позволяющий объединить эвристический метод оптимизации для задачи памяти, изложенный в работе [7], с описанным в подразделе 3.2 подходом разбиения АГВ на слои независимых вершин для сокращения затрат на синхронизацию (в частности, параллельное вычисление вершин).

Введем предикат $F : T \times T \to \{0, 1\}$:

$$F(t_1, t_2) = \begin{cases} 1, & \text{найдется слой, одновременно использующий } t_1 \text{ и } t_2 \\ 0, & \text{иначе} \end{cases}$$

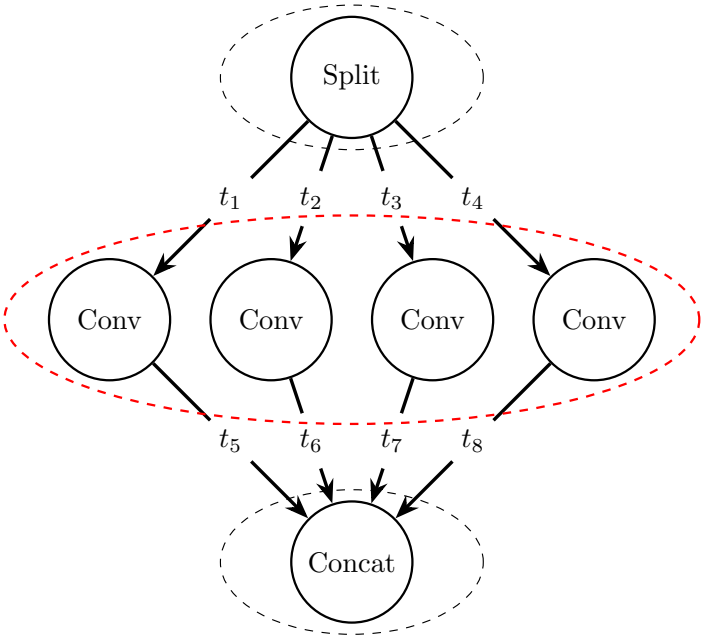
Использовав его, как дополнительное ограничение при проверке пересечения времен жизни тензоров, можно сформулировать приведенный алгоритм 2. Для выполнения алгоритма требуется предварительно найти времена жизни для всех тензоров, за это отвечает алгоритм 1.

При введении дополнительных ограничений может оказаться, что объем потребляемой памяти возрастает по сравнению с изначальным алгоритмом. Помимо этого имеется фактор среднего времени доступа в память ГП (чтение/запись), на который может влиять кэширование блоков памяти.

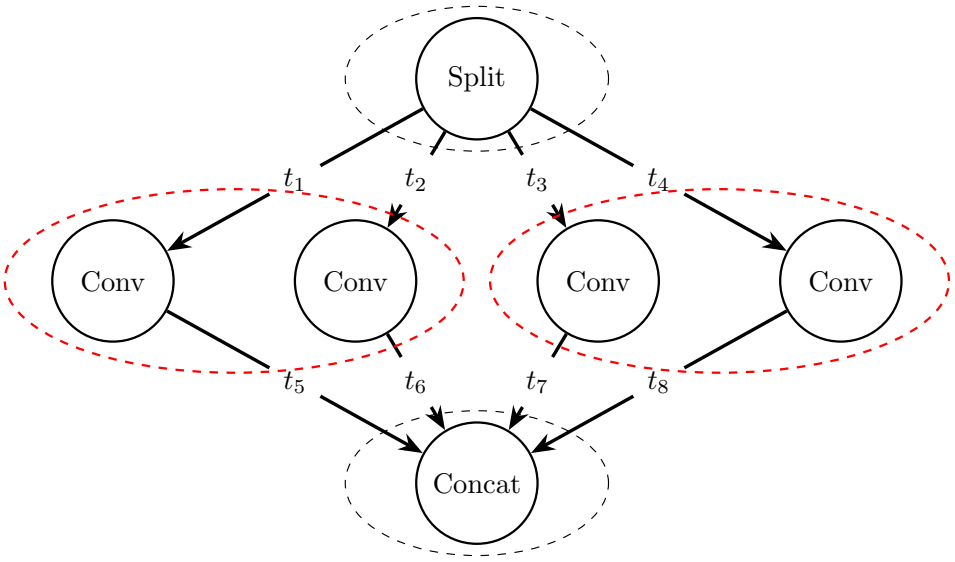
С целью варьирования дополнительных ограничений на тензоры, которые не могут использовать общую память, и количества точек синхронизации, предлагается дополнительно разбивать каждый слой АГВ на несколько непересекающихся *групп* одинакового размера (кроме, возможно, последней). Тогда вычисление отдельно взятого слоя будет представлять собой последовательное вычисление нескольких групп, между которыми будут установлены дополнительные точки синхронизации. Новые ограничения на совместное использование памяти тензорами зададим следующим образом:

$$F(t_1, t_2) = \begin{cases} 1, & \text{найдется группа, одновременно использующая } t_1 \text{ и } t_2 \\ 0, & \text{иначе} \end{cases}$$

Таким же образом полученный предикат $F(\cdot, \cdot)$ можно использовать в алгоритме 2.



(а)



(б)

Рис. 4: Пример применения предложенного метода с разбиением на одну группу (а) две группы (б)

Алгоритм 1 Нахождение времен жизни тензоров

функция CalculateTensorLifespans
    from $\leftarrow \varnothing$ ▷ ассоциативный массив (тензор, начало времени жизни)
    to $\leftarrow \varnothing$ ▷ ассоциативный массив (тензор, конец времени жизни)
    для всех $v_i \in V$ (**в порядке топологической сортировки**)
        для всех t — входящий/исходящий тензор для v_i
            если $t \notin$ from **то**
                from(t) $\leftarrow$ i
            to(t) $\leftarrow$ i
    lifespan $\leftarrow \{[\text{from}(t), \text{to}(t)] \mid t \in T\}$
    вернуть lifespan

Алгоритм 2 Эвристический поиск распределения памяти

функция CalculateTensorOffsets
    lifespan $\leftarrow$ CalculateTensorLifespans()
    offset_records $\leftarrow \varnothing$ ▷ множество пар (тензор, его сдвиг в памяти)
    для всех t — промежуточный тензор (**по убыванию** size(t))
        prev_offset $\leftarrow$ 0
        best_offset $\leftarrow$ null
        smallest_gap $\leftarrow \infty$
        ▷ пытаемся найти наименьший зазор для t
        для всех (x, offset) $\in$ offset_records **по возрастанию** offset
            если lifespan(t) $\cap$ lifespan(x) $\neq \varnothing$ **или** $F(t, x) = 1$ **то**
                если offset $\geqslant$ prev_offset **то**
                    gap $\leftarrow$ offset $-$ prev_offset
                    если size(t) $\leqslant$ gap **и** gap $<$ smallest_gap **то**
                        smallest_gap $\leftarrow$ gap
                        best_offset $\leftarrow$ prev_offset
                prev_offset $\leftarrow$ max(prev_offset, offset + size(x))
        ▷ если не нашли зазор, расширяем область памяти
        если best_offset = null **то**
            best_offset $\leftarrow$ prev_offset
        добавляем (t, best_offset) **в** offset_records
    вернуть offset_records

Планирование запуска АГВ

При теоретической оценке времени вычисления АГВ будем учитывать следующие факторы затрат:

1) Вычисление вершин/групп вершин

2) Синхронизации между вычислениями вершин/групп вершин

Время, затраченное на пересылку данных из оперативной памяти в память ГП и обратно, а также подготовку АГВ к вычислению, учитывать не будем.

Пусть в графе $G = (V, E)$ вершины занумерованы в порядке какой-то топологической сортировки. Введем обозначения: $\tau(v), v \in V$ — время вычисления вершины v, $s(v_i, v_{i+1}), v_i, v_{i+1} \in V$ — время, затраченное на синхронизацию между вершинами v_i и v_{i+1}.

В нашей модели вычислений суммарное затраченное время на вычисление графа G при базовой программной реализации:

$$\tau(G) = \sum_{i=1}^{n} \tau(v_i) + \sum_{i=1}^{n-1} s(v_i, v_{i+1})$$

Используем теперь построенное ранее разбиение на слои для более эффективного планирования вычислений. Рассмотрим разбиение на слои $\{L_0, \ldots, L_D\}$, и для каждого слоя $\tau(L)$ — время вычисления L, $s(L_i, L_{i+1})$ — время, затраченное на синхронизацию между слоями L_i и L_{i+1}. Тогда суммарное время вычисления графа G:

$$\tau'(G) = \sum_{i=0}^{D} \tau(L_i) + \sum_{i=1}^{D-1} s(L_i, L_{i+1})$$

В таком случае, теоретическое ускорение можно выразить так:

$$S(G) = \frac{\tau(G)}{\tau'(G)} = \frac{\sum\limits_{i=1}^{n} \tau(v_i) + \sum\limits_{i=1}^{n-1} s(v_i, v_{i+1})}{\sum\limits_{i=0}^{D} \tau(L_i) + \sum\limits_{i=1}^{D-1} s(L_i, L_{i+1})} \tag{4.1}$$

Поскольку при группировке вершин число синхронизаций уменьшается, а в силу роста степени параллелизма при объединении вершин в группы можно предположить, что $\tau(L_i) \leqslant \sum\limits_{v \in L_i} \tau(v) \Rightarrow S(G) \geqslant 1$. В разделе 5 будет протестировано соответствие действительности такой модели.

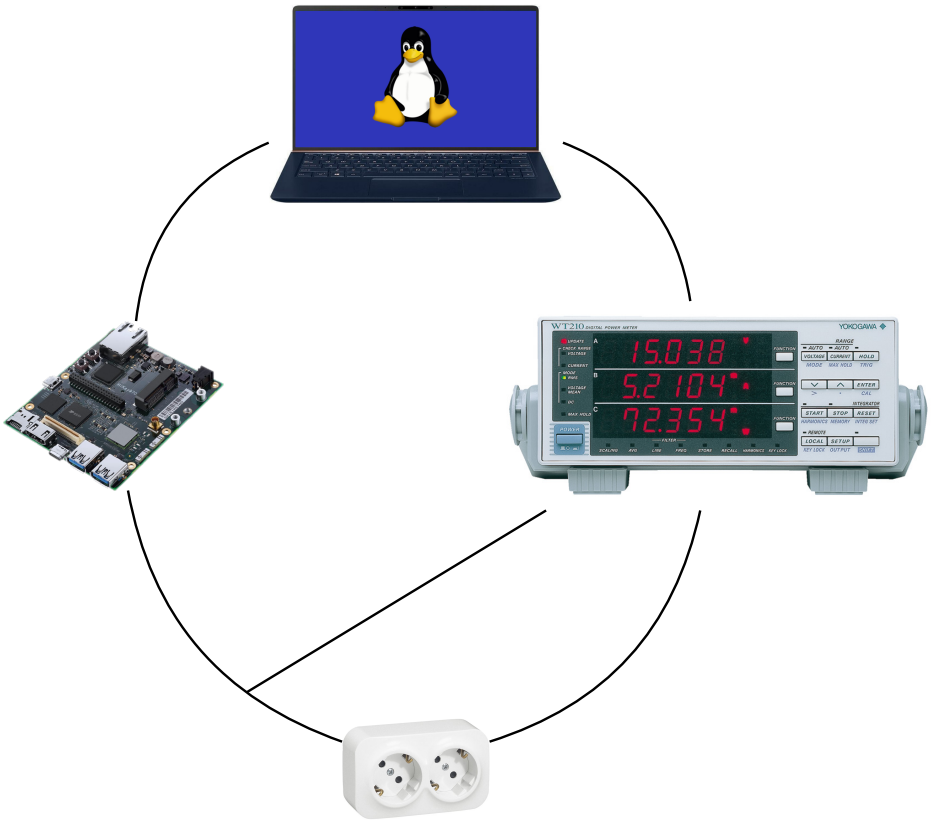


Рис. 5: Экспериментальная установка

5. Результаты экспериментов

В таблице 1 приведены полученные результаты замеров времени и памяти для некоторых моделей НС: ResNeXt [4], GoogLeNet [11], MobileNetV2 [12], Xception [13], InceptionV3 [14]. При конфигурации предложенного метода количество групп выбрано одинаковым для всех слоев и равняется либо 1, либо 2 (поскольку для всех сетей, кроме ResNeXt, размеры получаемых слоев не превосходят четырех вершин). Из этой таблицы видно, что для всех НС, кроме ResNeXt, оптимальное потребление памяти (по сравнению с изначальной эвристикой оптимизации памяти) достигается уже при одной группе, и разбиение на две группы также не дает улучшения по времени исполнения.

Представляет интерес более тщательный анализ экспериментов для ResNeXt (СНС с групповыми свертками размера 32). На Рис. 6 приведены полученные значения времени исполнения, объема используемой памяти, числа циклов ожидания обращений в память ГП и среднее потребление электрической энергии на запуск НС без учета потребления в покое. Оптимальный вы-

бор числа групп может быть нетривиален: например, исходя исключительно из времени исполнения, будет оптимально разбить каждый слой на две группы. Если взять также в расчет потребление памяти и электрической энергии, наилучшим значением может оказаться 16 или 8 групп. При разбиении на две группы ускорение составляет 6.85%, а потребление памяти сокращается на 72.4%.

Полученное сокращение времени вычислений при увеличении числа групп говорит о том, что модель 4.1 оказывается слишком упрощенной. А именно, она не учитывает, каким образом организован доступ в память ГП ее вычислительными ядрами. В нашем случае разбиение ResNeXt на две группы вместо одной дает относительное ускорение на 3.65%, также сокращается потребление памяти на 2.68%.

Приведенные измерения производились на плате HiKey970 с мобильным графическим процессором ARM Mali-G72 MP12, интегрированном в систему на кристалле HiSilicon Kirin 970. Реализация всех методов и экспериментов произведена в программно-аппаратной части библиотеки искусственного интеллекта MindSpore Lite с использованием языка C++ и интерфейса прикладного программирования Vulkan. Показатели ГП устройства были получены профилированием набором инструментов Arm Development Studio. Потребление электрической энергии было посчитано при помощи цифрового измерителя мощности Yokogawa WT210, показания с которого считывались утилитой командной строки Yokotool[2]. Визуально используемая экспериментальная установка изображена на Рис. 5.

[2]Исходный код `Yokotool`: `https://github.com/intel/yoko-tool`.

Таблица 1: Результаты экспериментов

Алгоритм		ResNeXt	GoogLeNet	MobileNetV2	Xception	InceptionV3
Базовая реализация	время, мс	234 ± 5	80 ± 3	81 ± 3	291 ± 2	369 ± 4
	память, Мб	393.7	67.9	78.3	268.5	198.1
Оптимизация памяти	время, мс	235 ± 6	82 ± 1	85 ± 4	293 ± 2	370 ± 3
	память, Мб	**108.8**	**42.4**	**30.3**	**113.2**	**122.6**
Послойное вычисление	время, мс	228 ± 2	**77 ± 1**	**80 ± 3**	**290 ± 2**	**361 ± 3**
	память, Мб	393.7	67.9	78.3	268.5	198.1
Разбиение на 1 группу	время, мс	227 ± 2	78 ± 2	82 ± 3	**290 ± 2**	**361 ± 4**
	память, Мб	111.8	**42.4**	**30.3**	**113.2**	**122.6**
Разбиение на 2 группы	время, мс	**219 ± 2**	78 ± 1	83 ± 4	291 ± 2	364 ± 3
	память, Мб	**108.8**	**42.4**	**30.3**	**113.2**	**122.6**

Для результатов замеров времени приведено:
среднее значение по выборке ± два среднеквадратичных отклонения

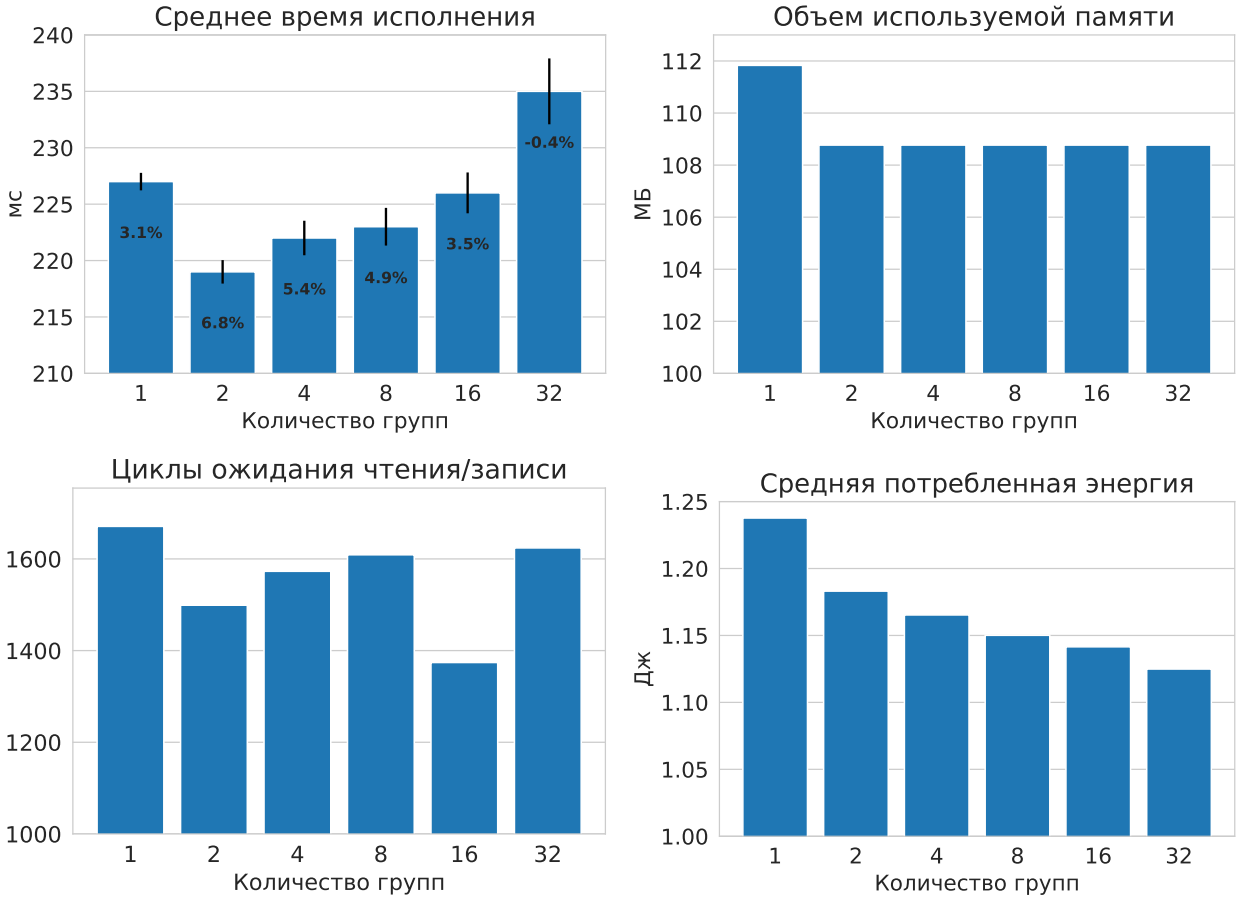


Рис. 6: Столбчатые диаграммы с результатами применения предложенного метода к сети ResNeXt

6. Заключение

В настоящей работе были рассмотрены задача оптимизации используемого объема памяти, а также задача параллельных вычислений нейронной сети. Предложен метод оптимизации на уровне ациклического графа вычислений, который совмещает эвристический алгоритм решения задачи оптимального распределения памяти с параллелизацией вычислений вершин графа. Его параметризация задает различные соотношения между потреблением времени, памяти и электрической энергии при вычислении НС на мобильном устройстве. Это позволяет разработчику выбрать компромисс, исходя из результатов измерений для различных конфигураций метода.

Предложенный метод был реализован в программно-аппаратной части библиотеки искусственного интеллекта MindSpore Lite на языке C++ с использованием интерфейса прикладного программирования Vulkan. Реализация протестирована на работоспособность и корректность на различных существующих НС.

Проведены эксперименты с использованием мобильного устройства с интегрированным графическим процессором, которые показывают применимость предложенного метода в том смысле, что он позволяет достигать сокращения потребляемого объема памяти и при этом уменьшать время вычисления НС. В частности, показано, что нет единой конфигурации параметров метода, позволяющей достигать оптимальных затрат для всех архитектур НС.

Результаты применения метода к сверточной НС с групповыми свертками ResNeXt [4] показывают, что при правильном выборе числа групп, на которые разбиваются слои графа, достигается ускорение на 6.85% и сокращение потребления памяти на 72.4%. Вместе с этим эмпирически показано, что на время вычисления предсказания НС существенное влияние могут оказывать, помимо синхронизаций между слоями, особенности обращений в память ГП вычислительными ядрами.

Таким образом, полностью выполнены задачи, поставленные в работе: исследованы существующие методы оптимизации объема памяти и затраченного времени на вычисление предсказания НС на уровне АГВ; разработан и реализован метод, позволяющий одновременно оптимизировать и затраченное время, и объем используемой памяти; эмпирически исследована применимость предложенного метода и произведено его сравнение с другими.

Список литературы

1. Object recognition with gradient-based learning / Y. LeCun [и др.] // Shape, contour and grouping in computer vision. — Springer, 1999. — с. 319—345.
2. *Zhang X.*, *Zhao J.*, *LeCun Y.* Character-level convolutional networks for text classification // Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1. — Montreal, Canada : MIT Press, 2015. — с. 649—657. — (NIPS'15).
3. *Kim Y.* Convolutional neural networks for sentence classification // Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. — 2014. — авг.
4. Aggregated residual transformations for deep neural networks / S. Xie [и др.] // Proceedings of the IEEE conference on computer vision and pattern recognition. — 2017. — с. 1492—1500.
5. *Krizhevsky A.*, *Sutskever I.*, *Hinton G. E.* Imagenet classification with deep convolutional neural networks // Advances in neural information processing systems. — 2012. — т. 25. — с. 1097—1105.
6. *Winograd S.* Arithmetic complexity of computations. т. 33. — Siam, 1980.
7. *Pisarchyk Y.*, *Lee J.* Efficient memory management for deep neural net inference // arXiv preprint arXiv:2001.03288. — 2020.
8. Node-level parallelization for deep neural networks with conditional independent graph / F. Zhou [и др.] // Neurocomputing. — 2017. — июнь. — т. 267.
9. *Garey M. R.*, *Johnson D. S.* Computers and intractability: a guide to the theory of NP-completeness. — USA : W. H. Freeman & Co., 1979. — с. 226.
10. Profile-guided memory optimization for deep neural networks / T. Sekiyama [и др.] // arXiv preprint arXiv:1804.10001. — 2018.

11. Going deeper with convolutions / C. Szegedy [и др.] // Proceedings of the IEEE conference on computer vision and pattern recognition. — 2015. — с. 1—9.

12. Mobilenetv2: Inverted residuals and linear bottlenecks / M. Sandler [и др.] // Proceedings of the IEEE conference on computer vision and pattern recognition. — 2018. — с. 4510—4520.

13. *Chollet F.* Xception: Deep learning with depthwise separable convolutions // Proceedings of the IEEE conference on computer vision and pattern recognition. — 2017. — с. 1251—1258.

14. Rethinking the inception architecture for computer vision / C. Szegedy [и др.] // Proceedings of the IEEE conference on computer vision and pattern recognition. — 2016. — с. 2818—2826.